Study guide: Approximation of functions

Hans Petter Langtangen[1,2]

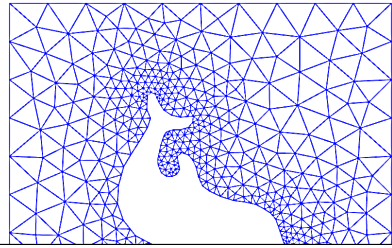
Center for Biomedical Computing, Simula Research Laboratory[1]

Department of Informatics, University of Oslo[2]

Sep 20, 2015

Finite elements can handle complex geometry, adaptive meshes, higher-order approximations and has a firm theory

- Can with ease solve PDEs in domains with *complex geometry*
- Can with ease create varying spatial resolution to get accuracy where it is needed
- Can with ease provide higher-order approximations
- Has a rigorous mathematical analysis framework



Solving PDEs by the finite element method

Stationary PDEs:

1. Transform the PDE problem to a *variational form*
2. Define function approximation over *finite elements*
3. Use a computational machinery to derive *linear systems*
4. Solve linear systems

Time-dependent PDEs:

- Finite elements *in space*
- Finite difference (or ODE solver) in time

We start with function approximation, then we treat PDEs

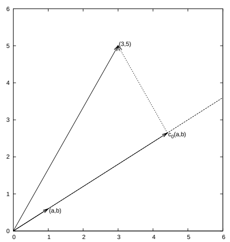
Learning strategy

- Start with approximation of functions, not PDEs
- Introduce finite element *approximations*
- See later how this machinery is applied to PDEs

Reason:

The finite element method has many concepts and a jungle of details. This learning strategy minimizes the mixing of ideas, concepts, and technical details.

Find a vector in some space that approximates a given vector in a space of higher dimension



The approximation is a linear combination of prescribed basis functions

General idea of finding an approximation $u(x)$ to some given $f(x)$:

$$u(x) = \sum_{i=0}^{N} c_i \psi_i(x)$$

where

- $\psi_i(x)$ are prescribed functions
- c_i, $i = 0, \ldots, N$ are unknown coefficients to be determined

We have three methods to determine the coefficients

We shall address three approaches:

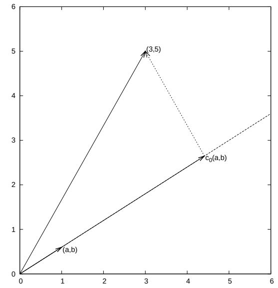
- The least squares method
- The projection (or Galerkin) method
- The interpolation (or collocation) method

Underlying motivation for our notation

Our mathematical framework for doing this is phrased in a way such that it becomes easy to understand and use the FEniCS software package for finite element computing.

Approximation of planar vectors; problem

Given a vector $\boldsymbol{f} = (3, 5)$, find an approximation to $\boldsymbol{f}$ directed along a given line.



Approximation of planar vectors; vector space terminology

$$V = \text{span}\,\{\psi_0\}$$

- ψ_0 is a basis vector in the space V
- Seek $\boldsymbol{u} = c_0\psi_0 \in V$
- Determine c_0 such that $\boldsymbol{u}$ is the "best" approximation to $\boldsymbol{f}$
- Visually, "best" is obvious

Define

- the error $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$
- the (Eucledian) scalar product of two vectors: $(\boldsymbol{u}, \boldsymbol{v})$
- the norm of $\boldsymbol{e}$: $||\boldsymbol{e}|| = \sqrt{(\boldsymbol{e}, \boldsymbol{e})}$

The least squares method; principle

- Idea: find c_0 such that $||\boldsymbol{e}||$ is minimized
- Mathematical convenience: minimize $E = ||\boldsymbol{e}||^2$

$$\frac{\partial E}{\partial c_0} = 0$$

The least squares method; calculations

$$\begin{aligned} E(c_0) &= (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - \boldsymbol{u}, \boldsymbol{f} - \boldsymbol{u}) = (\boldsymbol{f} - c_0\psi_0, \boldsymbol{f} - c_0\psi_0) \\ &= (\boldsymbol{f}, \boldsymbol{f}) - 2c_0(\boldsymbol{f}, \psi_0) + c_0^2(\psi_0, \psi_0) \end{aligned}$$

$$\frac{\partial E}{\partial c_0} = -2(\boldsymbol{f}, \psi_0) + 2c_0(\psi_0, \psi_0) = 0 \quad (1)$$

$$c_0 = \frac{(\boldsymbol{f}, \psi_0)}{(\psi_0, \psi_0)} = \frac{3a + 5b}{a^2 + b^2}$$

Observation to be used later: the vanishing derivative (1) can be alternatively written as

$$(\boldsymbol{e}, \psi_0) = 0$$

The projection (or Galerkin) method

- Last slide: min E is equivalent with $(\boldsymbol{e}, \psi_0) = 0$
- $(\boldsymbol{e}, \psi_0) = 0$ implies $(\boldsymbol{e}, \boldsymbol{v}) = 0$ for *any* $\boldsymbol{v} \in V$
- That is: instead of using the least-squares principle, we can require that $\boldsymbol{e}$ is orthogonal to *any* $\boldsymbol{v} \in V$
  (visually clear, but can easily be computed too)
- Precise math: find c_0 such that $(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V$
- Equivalent (see notes): find c_0 such that $(\boldsymbol{e}, \psi_0) = 0$
- Insert $\boldsymbol{e} = \boldsymbol{f} - c_0\psi_0$ and solve for c_0
- Same equation for c_0 and hence same solution as in the least squares method

Approximation of general vectors

Given a vector $\boldsymbol{f}$, find an approximation $\boldsymbol{u} \in V$:

$$V = \text{span}\,\{\psi_0, \ldots, \psi_N\}$$

We have a set of linearly independent basis vectors $\psi_0, \ldots, \psi_N$. Any $\boldsymbol{u} \in V$ can then be written as

$$\boldsymbol{u} = \sum_{j=0}^{N} c_j \psi_j$$

The least squares method

Idea: find $c_0, \ldots, c_N$ such that $E = ||\boldsymbol{e}||^2$ is minimized, $\boldsymbol{e} = \boldsymbol{f} - \boldsymbol{u}$.

$$\begin{aligned} E(c_0, \ldots, c_N) &= (\boldsymbol{e}, \boldsymbol{e}) = (\boldsymbol{f} - \sum_j c_j \psi_j, \boldsymbol{f} - \sum_j c_j \psi_j) \\ &= (\boldsymbol{f}, \boldsymbol{f}) - 2\sum_{j=0}^{N} c_j(\boldsymbol{f}, \psi_j) + \sum_{p=0}^{N}\sum_{q=0}^{N} c_p c_q (\psi_p, \psi_q) \end{aligned}$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = 0, \ldots, N$$

After some work we end up with a *linear system*

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N \tag{2}$$

$$A_{i,j} = (\psi_i, \psi_j) \tag{3}$$

The projection (or Galerkin) method

Can be shown that minimizing $||\boldsymbol{e}||$ implies that $\boldsymbol{e}$ is orthogonal to all $\boldsymbol{v} \in V$:

$$(\boldsymbol{e}, \boldsymbol{v}) = 0, \quad \forall \boldsymbol{v} \in V$$

which implies that $\boldsymbol{e}$ most be orthogonal to each basis vector:

$$(\boldsymbol{e}, \psi_i) = 0, \quad i = 0, \ldots, N$$

This orthogonality condition is the principle of the projection (or Galerkin) method. Leads to the same linear system as in the least squares method.

Approximation of a function in a function space

Let V be a *function space* spanned by a set of *basis functions* $\psi_0, \ldots, \psi_N$,

$$V = \text{span}\,\{\psi_0, \ldots, \psi_N\}$$

Find $u \in V$ as a linear combination of the basis functions:

$$u = \sum_{j \in \mathcal{I}_s} c_j \psi_j, \quad \mathcal{I}_s = \{0, 1, \ldots, N\}$$

The least squares method can be extended from vectors to functions

As in the vector case, minimize the (square) norm of the error, E, with respect to the coefficients c_j, $j \in \mathcal{I}_s$:

$$E = (e, e) = (f - u, f - u) = \left(f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), f(x) - \sum_{j \in \mathcal{I}_s} c_j \psi_j(x) \right)$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = \in \mathcal{I}_s$$

But what is the scalar product when ψ_i is a function?

$$(f, g) = \int_\Omega f(x) g(x)\, dx$$

(natural extension from Eucledian product $(\boldsymbol{u}, \boldsymbol{v}) = \sum_j u_j v_j$)

The least squares method; details

$$\begin{aligned} E(c_0, \ldots, c_N) &= (e, e) = (f - u, f - u) \\ &= (f, f) - 2\sum_{j \in \mathcal{I}_s} c_j (f, \psi_i) + \sum_{p \in \mathcal{I}_s}\sum_{q \in \mathcal{I}_s} c_p c_q (\psi_p, \psi_q) \end{aligned}$$

$$\frac{\partial E}{\partial c_i} = 0, \quad i = \in \mathcal{I}_s$$

The computations are *identical to the vector case*, and consequently we get a linear system

$$\sum_{j \in \mathcal{I}_s}^{N} A_{i,j} c_j = b_i, \; i \in \mathcal{I}_s, \quad A_{i,j} = (\psi_i, \psi_j), \; b_i = (f, \psi_i)$$

The projection (or Galerkin) method

As before, minimizing (e, e) is equivalent to

$$(e, \psi_i) = 0, \quad i \in \mathcal{I}_s$$

which is equivalent to

$$(e, v) = 0, \quad \forall v \in V$$

which is the projection (or Galerkin) method.

The algebra is the same as in the multi-dimensional vector case, and we get the same linear system as arose from the least squares method.

Example: fit a parabola by a straight line; problem

Problem

Approximate a parabola $f(x) = 10(x-1)^2 - 1$ by a straight line.

$$V = \text{span}\,\{1, x\}$$

That is, $\psi_0(x) = 1$, $\psi_1(x) = x$, and $N = 1$. We seek

$$u = c_0\psi_0(x) + c_1\psi_1(x) = c_0 + c_1 x$$

Example: fit a parabola by a straight line; solution

$$A_{0,0} = (\psi_0, \psi_0) = \int_1^2 1 \cdot 1\,dx = 1$$

$$A_{0,1} = (\psi_0, \psi_1) = \int_1^2 1 \cdot x\,dx = 3/2$$

$$A_{1,0} = A_{0,1} = 3/2$$

$$A_{1,1} = (\psi_1, \psi_1) = \int_1^2 x \cdot x\,dx = 7/3$$

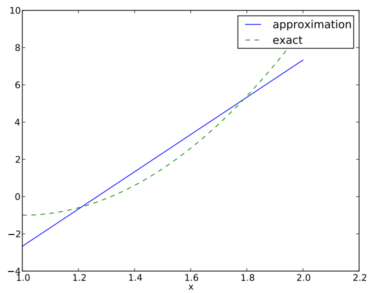
$$b_1 = (f, \psi_0) = \int_1^2 (10(x-1)^2 - 1) \cdot 1\,dx = 7/3$$

$$b_2 = (f, \psi_1) = \int_1^2 (10(x-1)^2 - 1) \cdot x\,dx = 13/3$$

Solution of 2x2 linear system:

$$c_0 = -38/3, \quad c_1 = 10, \quad u(x) = 10x - \frac{38}{3}$$

Example: fit a parabola by a straight line; plot



Ideas for implementing the least squares method via symbolic computations

Consider *symbolic computation* of the linear system, where

- $f(x)$ is given as a `sympy` expression `f` (involving the symbol `x`),
- `psi` is a list of $\{\psi_i\}_{i\in\mathcal{I}_s}$,
- `Omega` is a 2-tuple/list holding the domain Ω

Carry out the integrations, solve the linear system, and return $u(x) = \sum_j c_j\psi_j(x)$

Basic symbolic (SymPy) code for least squares

```
import sympy as sym

def least_squares(f, psi, Omega):
    N = len(psi) - 1
    A = sym.zeros((N+1, N+1))
    b = sym.zeros((N+1, 1))
    x = sym.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            A[i,j] = sym.integrate(psi[i]*psi[j],
                                   (x, Omega[0], Omega[1]))
            A[j,i] = A[i,j]
        b[i,0] = sym.integrate(psi[i]*f, (x, Omega[0], Omega[1]))
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i]
    return u, c
```

Observe: symmetric coefficient matrix so we can halve the integrations.

Improved code if symbolic integration fails

- If `sym.integrate` fails, it returns an `sym.Integral` object. We can test on this object and fall back on numerical integration.
- We can include a boolean argument `symbolic` to explicitly choose between symbolic and numerical computing.

```
def least_squares(f, psi, Omega, symbolic=True):
    ...
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = psi[i]*psi[j]
            if symbolic:
                I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
            if not symbolic or isinstance(I, sym.Integral):
                # Could not integrate symbolically,
                # fall back on numerical integration
                integrand = sym.lambdify([x], integrand)
                I = sym.mpmath.quad(integrand, [Omega[0], Omega[1]])
            A[i,j] = A[j,i] = I

        integrand = psi[i]*f
        if symbolic:
            I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sym.Integral):
            integrand = sym.lambdify([x], integrand)
```

Plotting of the solution

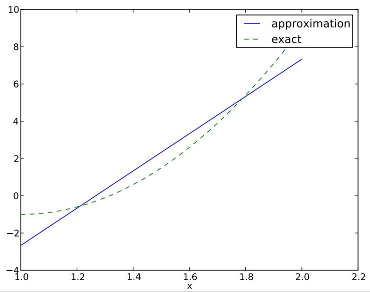
Compare f and u visually:

```
def comparison_plot(f, u, Omega, filename='tmp.pdf'):
    x = sym.Symbol('x')
    # Turn f and u to ordinary Python functions
    f = sym.lambdify([x], f, modules="numpy")
    u = sym.lambdify([x], u, modules="numpy")
    resolution = 401  # no of points in plot
    xcoor  = linspace(Omega[0], Omega[1], resolution)
    exact  = f(xcoor)
    approx = u(xcoor)
    plot(xcoor, approx)
    hold('on')
    plot(xcoor, exact)
    legend(['approximation', 'exact'])
    savefig(filename)
```

All code in module `approx1D.py`

Application of the software: fit a parabola by a straight line

```
>>> from approx1D import *
>>> x = sym.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x], Omega=[1, 2])
>>> comparison_plot(f, u, Omega=[1, 2])
```



The approximation is exact if $f \in V$

- What if we add $\psi_2 = x^2$ to the space V?
- That is, approximating a parabola by any parabola?
- (Hopefully we get the exact parabola!)

```
>>> from approx1D import *
>>> x = sym.Symbol('x')
>>> f = 10*(x-1)**2-1
>>> u, c = least_squares(f=f, psi=[1, x, x**2], Omega=[1, 2])
>>> print u
10*x**2 - 20*x + 9
>>> print sym.expand(f)
10*x**2 - 20*x + 9
```

The general result: perfect approximation if $f \in V$

- What if we use $\psi_i(x) = x^i$ for $i = 0, \ldots, N = 40$?
- The output from `least_squares` is $c_i = 0$ for $i > 2$

General result

If $f \in V$, least squares and projection/Galerkin give $u = f$.

Proof of why $f \in V$ gives exact u

If $f \in V$, $f = \sum_{j\in\mathcal{I}_s} d_j\psi_j$, for some $\{d_i\}_{i\in\mathcal{I}_s}$. Then

$$b_i = (f, \psi_i) = \sum_{j\in\mathcal{I}_s} d_j(\psi_j, \psi_i) = \sum_{j\in\mathcal{I}_s} d_j A_{i,j}$$

The linear system $\sum_j A_{i,j}c_j = b_i$, $i \in \mathcal{I}_s$, is then

$$\sum_{j\in\mathcal{I}_s} c_j A_{i,j} = \sum_{j\in\mathcal{I}_s} d_j A_{i,j}, \quad i \in \mathcal{I}_s$$

which implies that $c_i = d_i$ for $i \in \mathcal{I}_s$ and u is identical to f.

Finite-precision in numerical computations; question

The previous computations were symbolic. What if we solve the linear system numerically with standard arrays?

That is, f is parabola, but we approximate with

$$u(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_N x^N$$

We expect $c_2 = c_3 = \cdots = c_N = 0$ since $f \in V$ implies $u = f$.

Will we get this result with finite precision computer arithmetic?

Finite-precision in numerical computations; results

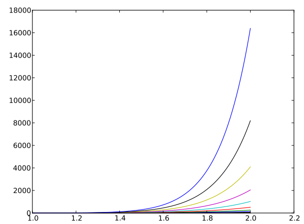
exact	sympy	numpy32	numpy64
9	9.62	5.57	8.98
-20	-23.39	-7.65	-19.93
10	17.74	-4.50	9.96
0	-9.19	4.13	-0.26
0	5.25	2.99	0.72
0	0.18	-1.21	-0.93
0	-2.48	-0.41	0.73
0	1.81	-0.013	-0.36
0	-0.66	0.08	0.11
0	0.12	0.04	-0.02
0	-0.001	-0.02	0.002

- Column 2: `matrix` and `lu_solve` from `sympy.mpmath.fp`
- Column 3: `numpy` matrix with 4-byte floats
- Column 4: `numpy` matrix with 8-byte floats

The ill-conditioning is due to almost linearly dependent basis functions for large N

- Significant round-off errors in the numerical computations (!)
- But if we plot the approximations they look good (!)

Source or problem: the x^i functions become almost linearly dependent as i grows:



Ill-conditioning: general conclusions

- Almost linearly dependent basis functions give almost singular matrices
- Such matrices are said to be *ill conditioned*, and Gaussian elimination is severely affected by round-off errors
- The basis $1, x, x^2, x^3, x^4, \ldots$ is a bad basis
- Polynomials are fine as basis, but the more orthogonal they are, $(\psi_i, \psi_j) \approx 0$, the better

Fourier series approximation; problem and code

Let's approximate f by a typical Fourier series expansion

$$u(x) = \sum_i a_i \sin i\pi x = \sum_{j=0}^{N} c_j \sin((j+1)\pi x)$$

which means that

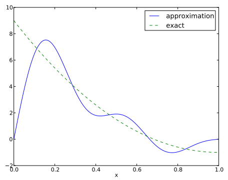
$$V = \text{span}\left\{\sin \pi x, \sin 2\pi x, \ldots, \sin(N+1)\pi x\right\}$$

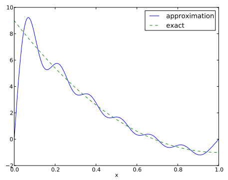
Computations using the `least_squares` function:

```
N = 3
from sympy import sin, pi
psi = [sin(pi*(i+1)*x) for i in range(N+1)]
f = 10*(x-1)**2 - 1
Omega = [0, 1]
u, c = least_squares(f, psi, Omega)
comparison_plot(f, u, Omega)
```

Fourier series approximation; plot

Left: $N = 3$, right: $N = 11$:





Problem:

All $\psi_i(0) = 0$ and hence $u(0) = 0 \neq f(0) = 9$. Similar problem at $x = 1$. The boundary values of u are always wrong!

Fourier series approximation; improvements

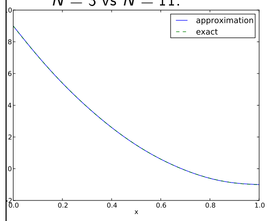
- Considerably improvement with $N = 11$, but still undesired discrepancy at $x = 0$ and $x = 1$
- Possible remedy: add a term that leads to correct boundary values

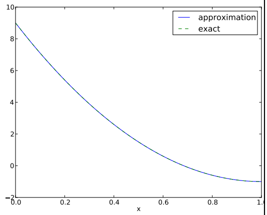
$$u(x) = f(0)(1 - x) + xf(1) + \sum_{j\in\mathcal{I}_s} c_j\psi_j(x)$$

The extra terms ensure $u(0) = f(0)$ and $u(1) = f(1)$ and is a strikingly good help to get a good approximation!

Fourier series approximation; final results

$N = 3$ vs $N = 11$:





Orthogonal basis functions

This choice of sine functions as basis functions is popular because

- the basis functions are orthogonal: $(\psi_i, \psi_j) = 0$
- implying that $A_{i,j}$ is a diagonal matrix
- implying that we can solve for $c_i = 2\int_0^1 f(x)\sin((i+1)\pi x)dx$
- and what we get is the standard Fourier sine series of f

In general, for an orthogonal basis, $A_{i,j}$ is diagonal and we can easily solve for c_i:

$$c_i = \frac{b_i}{A_{i,i}} = \frac{(f, \psi_i)}{(\psi_i, \psi_i)}$$

Function for the least squares method with orthogonal basis functions

```
def least_squares_orth(f, psi, Omega):
    N = len(psi) - 1
    A = [0]*(N+1)
    b = [0]*(N+1)
    x = sym.Symbol('x')
    for i in range(N+1):
        A[i] = sym.integrate(psi[i]**2, (x, Omega[0], Omega[1]))
        b[i] = sym.integrate(psi[i]*f,  (x, Omega[0], Omega[1]))
    c = [b[i]/A[i] for i in range(len(b))]
    u = 0
    for i in range(len(psi)):
        u += c[i]*psi[i]
    return u, c
```

Function for the least squares method with orthogonal basis functions; symbolic and numerical integration

Extensions:

- We can choose between symbolic or numerical integration (`symbolic` argument).
- If symbolic, we fall back on numerical integration after failure (`sym.Integral` is returned from `sym.integrate`).

```
def least_squares_orth(f, psi, Omega, symbolic=True):
    ...
    for i in range(N+1):
        # Diagonal matrix term
        A[i] = sym.integrate(psi[i]**2, (x, Omega[0], Omega[1]))

        # Right-hand side term
        integrand = psi[i]*f
        if symbolic:
            I = sym.integrate(integrand,  (x, Omega[0], Omega[1]))
        if not symbolic or isinstance(I, sym.Integral):
            print 'numerical integration of', integrand
            integrand = sym.lambdify([x], integrand)
            I = sym.mpmath.quad(integrand, [Omega[0], Omega[1]])
        b[i] = I
    ...
```

The collocation or interpolation method; ideas and math

Here is another idea for approximating $f(x)$ by $u(x) = \sum_j c_j\psi_j$:

- Force $u(x_i) = f(x_i)$ at some selected *collocation* points $\{x_i\}_{i\in\mathcal{I}_s}$
- Then u is said to *interpolate* f
- The method is known as *interpolation* or *collocation*

$$u(x_i) = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x_i) = f(x_i) \quad i \in \mathcal{I}_s, N$$

This is a linear system with no need for integration:

$$\sum_{j\in\mathcal{I}_s} A_{i,j}c_j = b_i, \quad i \in \mathcal{I}_s \tag{5}$$

$$A_{i,j} = \psi_j(x_i) \tag{6}$$

$$b_i = f(x_i) \tag{7}$$

No symmetric matrix: $\psi_j(x_i) \neq \psi_i(x_j)$ in general

The collocation or interpolation method; implementation

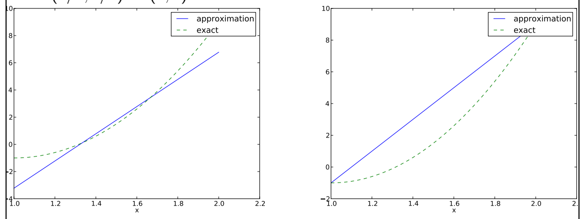
`points` holds the interpolation/collocation points

```python
def interpolation(f, psi, points):
    N = len(psi) - 1
    A = sym.zeros((N+1, N+1))
    b = sym.zeros((N+1, 1))
    x = sym.Symbol('x')
    # Turn psi and f into Python functions
    psi = [sym.lambdify([x], psi[i]) for i in range(N+1)]
    f = sym.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            A[i,j] = psi[j](points[i])
        b[i,0] = f(points[i])
    c = A.LUsolve(b)
    u = 0
    for i in range(len(psi)):
        u += c[i,0]*psi[i](x)
    return u
```

The collocation or interpolation method; approximating a parabola by linear functions

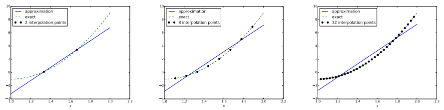
- Potential difficulty: how to choose x_i?
- The results are sensitive to the points!

$(4/3, 5/3)$ vs $(1, 2)$:



The regression method

- Idea: Interpolation (collocation) method, but use $m \gg N+1$ points
- Problem: More equations than unknowns
- But this is well known as regression in statistics



The regression method leads to an overdetermined linear system

Overdetermined linear system:

$$u(x_i) = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x_i) = f(x_i), \quad i = 0, 1, \ldots, m$$

$$\sum_{j\in\mathcal{I}_s} A_{i,j}c_j = b_i, \quad i = 0, 1, \ldots, m$$

$$A_{i,j} = \psi_j(x_i), \quad b_i = f(x_i)$$

A least squares method is used to solve overdetermined linear systems

- Cannot (in general) solve $Ac = b$ when there are more equations than unknowns
- Idea: Minimize $r = b - Ac$ instead
- Result: the *normal equations* $A^TAc = A^Tb$
- $(N+1)\times(N+1)$ system
- Write the normal equations as $Bc = d$

$$B_{i,j} = \sum_k A^Ti, kA_{k,j} = \sum_k Ak, iA_{k,j} = \sum_{k=0}^m \psi_i(x_k\psi_j(x_k$$

$$d_i = \sum_k A^T_{i,k}b_k = \sum_k A_{k,i}b_k = \sum_{k=0}^m \psi_i(x_k)f(x_k)$$

Implementation

```python
def regression(f, psi, points):
    N = len(psi) - 1
    m = len(points)
    # Use numpy arrays and numerical computing
    B = np.zeros((N+1, N+1))
    d = np.zeros(N+1)
    # Wrap psi and f in Python functions rather than expressions
    # so that we can evaluate psi at points[i]
    x = sym.Symbol('x')
    psi_sym = psi  # save symbolic expression
    psi = [sym.lambdify([x], psi[i]) for i in range(N+1)]
    f = sym.lambdify([x], f)
    for i in range(N+1):
        for j in range(N+1):
            B[i,j] = 0
            for k in range(m+1):
                B[i,j] += psi[i](points[k])*psi[j](points[k])
        d[i] = 0
        for k in range(m+1):
            d[i] += psi[i](points[k])*f(points[k])
    c = np.linalg.solve(B, d)
    u = sum(c[i]*psi_sym[i] for i in range(N+1))
    return u, c
```

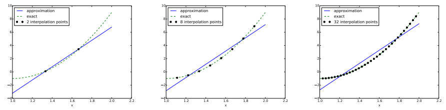
Example on using the regression method; code

- Approximate $f(x) = 10(x-1)^2 - 1$ by a linear function on $\Omega = [1, 2]$

```
import sympy as sym
x = sym.Symbol('x')
f = 10*(x-1)**2 - 1
psi = [1, x]
Omega = [1, 2]
m_values = [2-1, 8-1, 64-1]
# Create m+3 points and use the inner m+1 points
for m in m_values:
    points = np.linspace(Omega[0], Omega[1], m+3)[1:-1]
    u, c = regression(f, psi, points)
    comparison_plot(f, u, Omega, points=points,
        points_legend='%d interpolation points' % (m+1))
```

Example on using the regression method; result

$$u(x) = 10x - 13.2, \quad 2 \text{ points}$$
$$u(x) = 10x - 12.7, \quad 8 \text{ points}$$
$$u(x) = 10x - 12.7, \quad 64 \text{ points}$$



What is the regression method used for?

- It is one of the most dominating methods for approximating data in statistics
- Not so common for approximating functions
- Not much used for solving differential equations
- Recently very popular for statistical uncertainty quantification: approximating the mapping from input parameters to the solution via polynomials and the regression method (called polynomial chaos expansions)

Lagrange polynomials; motivation and ideas

Motivation:

- The interpolation/collocation method avoids integration
- With a diagonal matrix $A_{i,j} = \psi_j(x_i)$ we can solve the linear system by hand

The *Lagrange interpolating polynomials* ψ_j have the property that

$$\psi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Hence, $c_i = f(x_i)$ and

$$u(x) = \sum_{j\in\mathcal{I}_s} f(x_i)\psi_i(x)$$

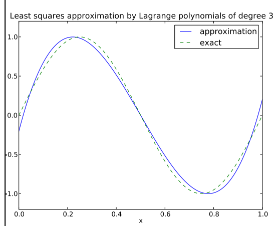
- Lagrange polynomials and interpolation/collocation look convenient

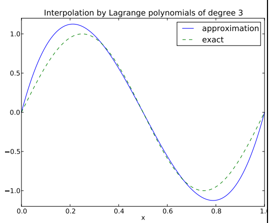
Lagrange polynomials; formula and code

$$\psi_i(x) = \prod_{j=0,j\neq i}^{N} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_N}{x_i - x_N}$$

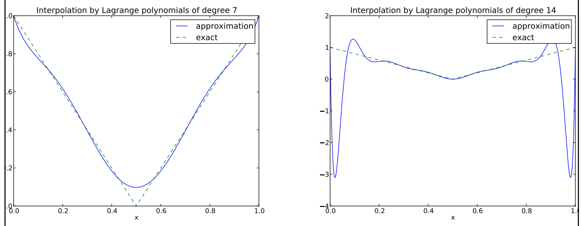
```
def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p
```

Lagrange polynomials; successful example



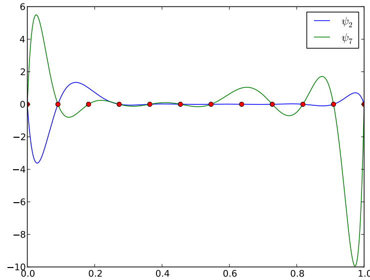


Lagrange polynomials; a less successful example



Lagrange polynomials; oscillatory behavior

12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



Problem: strong oscillations near the boundaries for larger N values

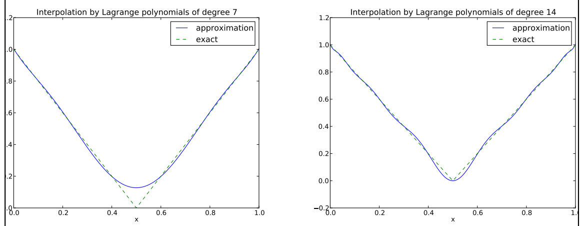
Lagrange polynomials; remedy for strong oscillations

The oscillations can be reduced by a more clever choice of interpolation points, called the *Chebyshev nodes*:

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left(\frac{2i+1}{2(N+1)}pi\right), \quad i = 0\ldots,N$$

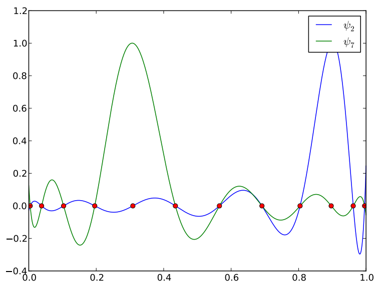
on an interval $[a,b]$.

Lagrange polynomials; recalculation with Chebyshev nodes

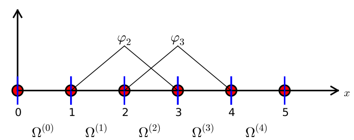


Lagrange polynomials; less oscillations with Chebyshev nodes

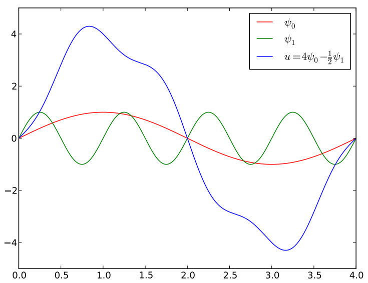
12 points, degree 11, plot of two of the Lagrange polynomials - note that they are zero at all points except one.



Finite element basis functions

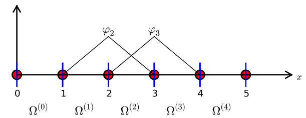


The basis functions have so far been global: $\psi_i(x) \neq 0$ almost everywhere

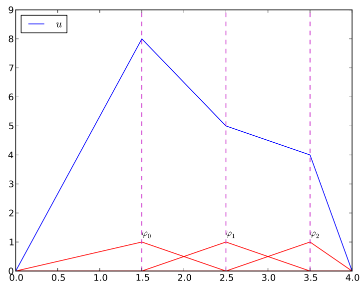


In the finite element method we use basis functions with local support

- *Local support*: $\psi_i(x) \neq 0$ for x in a small subdomain of Ω
- Typically hat-shaped
- $u(x)$ based on these ψ_i is a piecewise polynomial defined over many (small) subdomains
- We introduce φ_i as the name of these finite element hat functions (and for now choose $\psi_i = \varphi_i$)



The linear combination of hat functions is a piecewise linear function



Elements and nodes

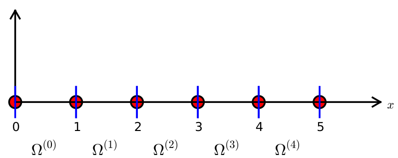
Split Ω into N_e non-overlapping subdomains called *elements*:

$$\Omega = \Omega^{(0)} \cup \cdots \cup \Omega^{(N_e)}$$

On each element, introduce N_n points called *nodes*: $x_0, \ldots, x_{N_n-1}$

- The finite element basis functions are named $\varphi_i(x)$
- $\varphi_i = 1$ at node i and 0 at all other nodes
- φ_i is a Lagrange polynomial on each element
- For nodes at the boundary between two elements, φ_i is made up of a Lagrange polynomial over each element

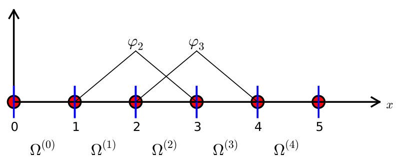
Example on elements with two nodes (P1 elements)



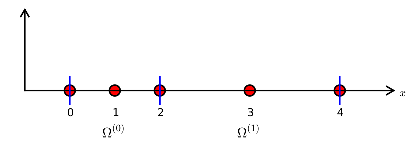
Data structure: `nodes` holds coordinates or nodes, `elements` holds the node numbers in each element

```
nodes = [0, 1.2, 2.4, 3.6, 4.8, 5]
elements = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5]]
```

Illustration of two basis functions on the mesh

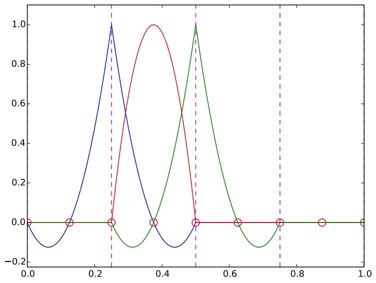


Example on elements with three nodes (P2 elements)

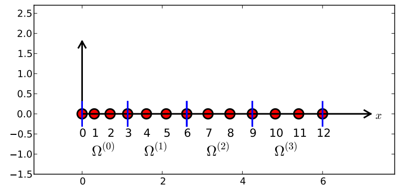



```
nodes = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0]
elements = [[0, 1, 2], [2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

Some corresponding basis functions (P2 elements)

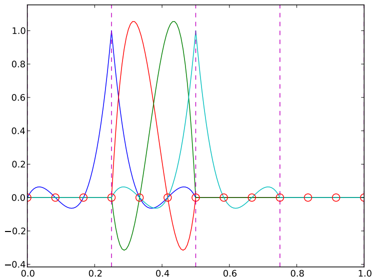


Examples on elements with four nodes (P3 elements)

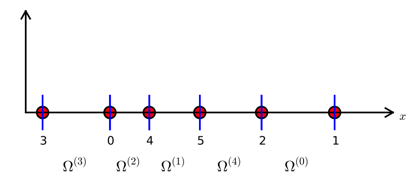



```
d = 3  # d+1 nodes per element
num_elements = 4
num_nodes = num_elements*d + 1
nodes = [i*0.5 for i in range(num_nodes)]
elements = [[i*d+j for j in range(d+1)] for i in range(num_elements)]
```

Some corresponding basis functions (P3 elements)



The numbering does not need to be regular from left to right




```
nodes = [1.5, 5.5, 4.2, 0.3, 2.2, 3.1]
elements = [[2, 1], [4, 5], [0, 4], [3, 0], [5, 2]]
```

Interpretation of the coefficients c_i

Important property: c_i is the value of u at node i, x_i:

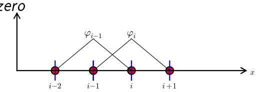
$$u(x_i) = \sum_{j\in\mathcal{I}_s} c_j\varphi_j(x_i) = c_i\varphi_i(x_i) = c_i$$

because $\varphi_j(x_i) = 0$ if $i \neq j$ and $\varphi_i(x_i) = 1$

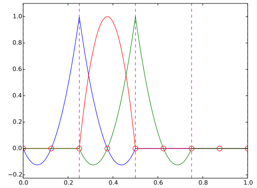
Properties of the basis functions

- $\varphi_i(x) \neq 0$ only on those elements that contain global node i
- $\varphi_i(x)\varphi_j(x) \neq 0$ if and only if i and j are global node numbers in the same element

Since $A_{i,j} = \int \varphi_i\varphi_j \,\mathrm{d}x$, *most of the elements in the coefficient matrix will be zero*

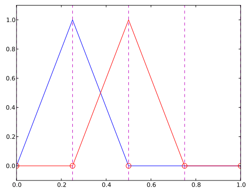


How to construct quadratic φ_i (P2 elements)



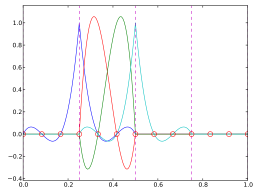
1. Associate Lagrange polynomials with the nodes in an element
2. When the polynomial is 1 on the element boundary, combine it with the polynomial in the neighboring element that is also 1 at the same point

Example on linear φ_i (P1 elements)

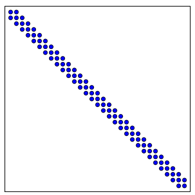


$$\varphi_i(x) = \begin{cases} 0, & x < x_{i-1} \\ (x - x_{i-1})/h & x_{i-1} \le x < x_i \\ 1 - (x - x_i)/h, & x_i \le x < x_{i+1} \\ 0, & x \ge x_{i+1} \end{cases}$$

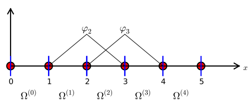
Example on cubic φ_i (P3 elements)



Calculating the linear system for c_i



Computing a specific matrix entry (1)

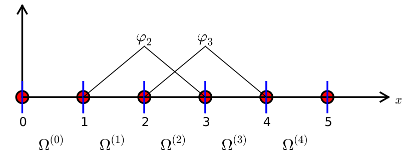


$A_{2,3} = \int_\Omega \varphi_2\varphi_3 dx$: $\varphi_2\varphi_3 \neq 0$ only over element 2. There,

$$\varphi_3(x) = (x - x_2)/h, \quad \varphi_2(x) = 1 - (x - x_2)/h$$

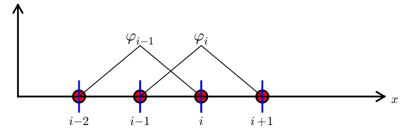
$$A_{2,3} = \int_\Omega \varphi_2\varphi_3 \,\mathrm{d}x = \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right) \frac{x - x_2}{h} \,\mathrm{d}x = \frac{h}{6}$$

Computing a specific matrix entry (2)



$$A_{2,2} = \int_{x_1}^{x_2} \left(\frac{x - x_1}{h}\right)^2 \mathrm{d}x + \int_{x_2}^{x_3} \left(1 - \frac{x - x_2}{h}\right)^2 \mathrm{d}x = \frac{2h}{3}$$

Calculating a general row in the matrix; figure



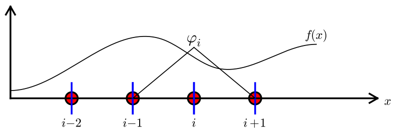
$$A_{i,i-1} = \int_\Omega \varphi_i \varphi_{i-1} \,\mathrm{d}x = ?$$

Calculating a general row in the matrix; details

$$\begin{aligned} A_{i,i-1} &= \int_\Omega \varphi_i \varphi_{i-1} \,\mathrm{d}x \\ &= \underbrace{\int_{x_{i-2}}^{x_{i-1}} \varphi_i \varphi_{i-1} \,\mathrm{d}x}_{\varphi_i = 0} + \int_{x_{i-1}}^{x_i} \varphi_i \varphi_{i-1} \,\mathrm{d}x + \underbrace{\int_{x_i}^{x_{i+1}} \varphi_i \varphi_{i-1} \,\mathrm{d}x}_{\varphi_{i-1} = 0} \\ &= \int_{x_{i-1}}^{x_i} \underbrace{\left(\frac{x - x_i}{h}\right)}_{\varphi_i(x)} \underbrace{\left(1 - \frac{x - x_{i-1}}{h}\right)}_{\varphi_{i-1}(x)} \mathrm{d}x = \frac{h}{6} \end{aligned}$$

- $A_{i,i+1} = A_{i,i-1}$ due to symmetry
- $A_{i,i} = 2h/3$ (same calculation as for $A_{2,2}$)
- $A_{0,0} = A_{N,N} = h/3$ (only one element)

Calculation of the right-hand side



$$b_i = \int_\Omega \varphi_i(x) f(x) \,\mathrm{d}x = \int_{x_{i-1}}^{x_i} \frac{x - x_{i-1}}{h} f(x) \,\mathrm{d}x + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x - x_i}{h}\right) f(x)$$

Need a specific $f(x)$ to do more...

Specific example with two elements; linear system and solution

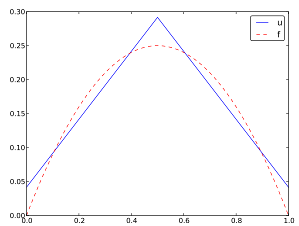
- $f(x) = x(1 - x)$ on $\Omega = [0, 1]$
- Two equal-sized elements $[0, 0.5]$ and $[0.5, 1]$

$$A = \frac{h}{6} \begin{pmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{pmatrix}, \quad b = \frac{h^2}{12} \begin{pmatrix} 2 - 3h \\ 12 - 14h \\ 10 - 17h \end{pmatrix}$$

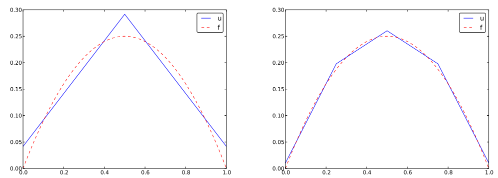
$$c_0 = \frac{h^2}{6}, \quad c_1 = h - \frac{5}{6}h^2, \quad c_2 = 2h - \frac{23}{6}h^2$$

Specific example with two elements; plot

$$u(x) = c_0 \varphi_0(x) + c_1 \varphi_1(x) + c_2 \varphi_2(x)$$



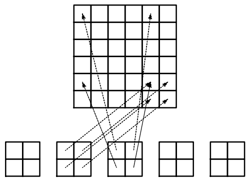
Specific example with four elements; plot



Specific example: what about P2 elements?

Recall: if $f \in V$, u becomes exact. When f is a parabola, any choice of P2 elements (1 or many) will give $u = f$ exactly. The same is true for P3, P4, ... elements since all of them can represent a 2nd-degree polynomial exactly.

Assembly of elementwise computations

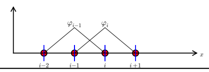


Split the integrals into elementwise integrals

$$A_{i,j} = \int_\Omega \varphi_i\varphi_j dx = \sum_e \int_{\Omega^{(e)}} \varphi_i\varphi_j dx, \quad A^{(e)}_{i,j} = \int_{\Omega^{(e)}} \varphi_i\varphi_j dx$$

Important observations:

- $A^{(e)}_{i,j} \neq 0$ if and only if i and j are nodes in element e (otherwise no overlap between the basis functions)
- All the nonzero elements in $A^{(e)}_{i,j}$ are collected in an *element matrix*
- The element matrix has contributions from the φ_i functions associated with the nodes in element
- It is convenient to introduce a *local numbering* of the nodes in an element: $0, 1, \ldots, d$



The element matrix and local vs global node numbers

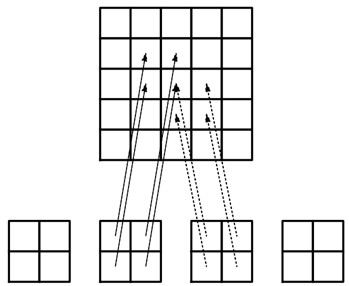
$$\tilde{A}^{(e)} = \{\tilde{A}^{(e)}_{r,s}\}, \quad \tilde{A}^{(e)}_{r,s} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}\varphi_{q(e,s)} dx, \quad r, s \in I_d = \{0, \ldots, d\}$$

Now,

- r, s run over *local node numbers* in an element: $0, 1, \ldots, d$
- i, j run over *global node numbers* $i, j \in \mathcal{I}_s = \{0, 1, \ldots, N\}$
- $i = q(e, r)$: mapping of local node number r in element e to the global node number i (math equivalent to `i=elements[e][r]`)
- Add $\tilde{A}^{(e)}_{r,s}$ into the global $A_{i,j}$ (*assembly*)

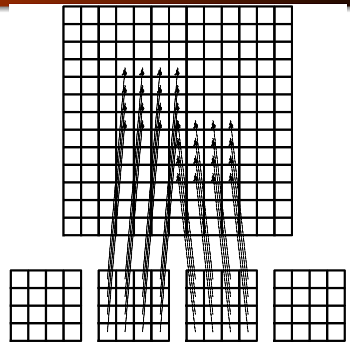
$$A_{q(e,r),q(e,s)} := A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad r, s \in I_d$$

Illustration of the matrix assembly: regularly numbered P1 elements



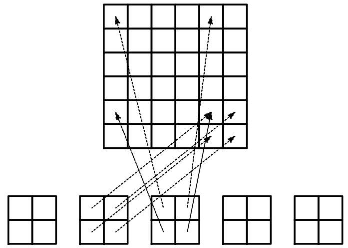
Animation

Illustration of the matrix assembly: regularly numbered P3 elements



Animation

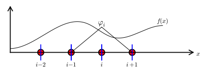
Illustration of the matrix assembly: irregularly numbered P1 elements



Animation

Assembly of the right-hand side

$$b_i = \int_\Omega f(x)\varphi_i(x)dx = \sum_e \int_{\Omega^{(e)}} f(x)\varphi_i(x)dx, \quad b_i^{(e)} = \int_{\Omega^{(e)}} f(x)\varphi_i(x)dx$$



Important observations:

- $b_i^{(e)} \neq 0$ if and only if global node i is a node in element e (otherwise $\varphi_i = 0$)
- The $d+1$ nonzero $b_i^{(e)}$ can be collected in an *element vector* $\tilde{b}_r^{(e)} = \{\tilde{b}_r^{(e)}\}$, $r \in I_d$

Assembly:

$$b_{q(e,r)} := b_{q(e,r)} + \tilde{b}_r^{(e)}, \quad r, s \in I_d$$

Mapping to a reference element

Instead of computing

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{x_L}^{x_R} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx$$

we now map $[x_L, x_R]$ to a standardized reference element domain $[-1, 1]$ with local coordinate X

We use affine mapping: linear stretch of $X \in [-1, 1]$ to $x \in [x_L, x_R]$

$$x = \frac{1}{2}(x_L + x_R) + \frac{1}{2}(x_R - x_L)X$$

or rewritten as

$$x = x_m + \frac{1}{2}hX, \qquad x_m = (x_L + x_R)/2, \quad h = x_R - x_L$$

Integral transformation

Reference element integration: just change integration variable from x to X. Introduce local basis function

$$\tilde{\varphi}_r(X) = \varphi_{q(e,r)}(x(X))$$

$$\tilde{A}_{r,s}^{(e)} = \int_{\Omega^{(e)}} \varphi_{q(e,r)}(x)\varphi_{q(e,s)}(x)dx = \int_{-1}^{1} \tilde{\varphi}_r(X)\tilde{\varphi}_s(X) \underbrace{\frac{dx}{dX}}_{\det J = h/2} dX = \int_{-1}^{1} \tilde{\varphi}_r$$

$$\tilde{b}_r^{(e)} = \int_{\Omega^{(e)}} f(x)\varphi_{q(e,r)}(x)dx = \int_{-1}^{1} f(x(X))\tilde{\varphi}_r(X)\det J\, dX$$

Advantages of the reference element

- Always the same domain for integration: $[-1, 1]$
- We only need formulas for $\tilde{\varphi}_r(X)$ over one element (no *piecewise* polynomial definition)
- $\tilde{\varphi}_r(X)$ is the same for all elements: no dependence on element length and location, which is "factored out" in the mapping and $\det J$

Standardized basis functions for P1 elements

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X) \tag{8}$$

$$\tilde{\varphi}_1(X) = \frac{1}{2}(1 + X) \tag{9}$$

Note: simple polynomial expressions (no need to consider piecewisely defined functions)

Standardized basis functions for P2 elements

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X - 1)X \tag{10}$$

$$\tilde{\varphi}_1(X) = 1 - X^2 \tag{11}$$

$$\tilde{\varphi}_2(X) = \frac{1}{2}(X + 1)X \tag{12}$$

Easy to generalize to arbitrary order!

How to find the polynomial expressions?

Three alternatives:

1. Map the global basis function $\varphi_i(x)$ over an element to X coordinates
2. Compute $\tilde{\varphi}_r(X)$ from scratch using
   1. a given polynomial order d
   2. $\tilde{\varphi}_r(X) = 1$ at local node 1
   3. $\tilde{\varphi}_r(X) = 1$ at all other local nodes
3. Use formulas for Lagrange interpolating polynomials on the element

Integration over a reference element; element matrix

P1 elements and $f(x) = x(1 - x)$.

$$\begin{aligned}
\tilde{A}^{(e)}_{0,0} &= \int_{-1}^{1} \tilde{\varphi}_0(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1 - X)\frac{1}{2}(1 - X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1 - X)^2 dX = \frac{h}{3}
\end{aligned} \tag{13}$$

$$\begin{aligned}
\tilde{A}^{(e)}_{1,0} &= \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} \frac{1}{2}(1 + X)\frac{1}{2}(1 - X)\frac{h}{2}dX = \frac{h}{8}\int_{-1}^{1}(1 - X^2) dX = \frac{h}{6}
\end{aligned} \tag{14}$$

$$\tilde{A}^{(e)}_{0,1} = \tilde{A}^{(e)}_{1,0} \tag{15}$$

$$\tilde{A}^{(e)}_{1,1} = \int_{-1}^{1} \tilde{\varphi}_1(X)\tilde{\varphi}_1(X)\frac{h}{2}dX$$

Integration over a reference element; element vector

$$\begin{aligned}
\tilde{b}^{(e)}_0 &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_0(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1 - X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 + \frac{1}{6}h^2 x_m - \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m
\end{aligned} \tag{17}$$

$$\begin{aligned}
\tilde{b}^{(e)}_1 &= \int_{-1}^{1} f(x(X))\tilde{\varphi}_1(X)\frac{h}{2}dX \\
&= \int_{-1}^{1} (x_m + \frac{1}{2}hX)(1 - (x_m + \frac{1}{2}hX))\frac{1}{2}(1 + X)\frac{h}{2}dX \\
&= -\frac{1}{24}h^3 - \frac{1}{6}h^2 x_m + \frac{1}{12}h^2 - \frac{1}{2}hx_m^2 + \frac{1}{2}hx_m
\end{aligned} \tag{18}$$

x_m: element midpoint.

Tedious calculations! Let's use symbolic software

```
>>> import sympy as sym
>>> x, x_m, h, X = sym.symbols('x x_m h X')
>>> sym.integrate(h/8*(1-X)**2, (X, -1, 1))
h/3
>>> sym.integrate(h/8*(1+X)*(1-X), (X, -1, 1))
h/6
>>> x = x_m + h/2*X
>>> b_0 = sym.integrate(h/4*x*(1-x)*(1-X), (X, -1, 1))
>>> print b_0
-h**3/24 + h**2*x_m/6 - h**2/12 - h*x_m**2/2 + h*x_m/2
```

Can print out in LaTeX too (convenient for copying into reports):

```
>>> print sym.latex(b_0, mode='plain')
- \frac{1}{24} h^{3} + \frac{1}{6} h^{2} x_{m}
- \frac{1}{12} h^{2} - \half h x_{m}^{2}
+ \half h x_{m}
```

Implementation

- Coming functions appear in `fe_approx1D.py`
- Functions can operate in symbolic or numeric mode
- The code documents all steps in finite element calculations!

Compute finite element basis functions in the reference element

Let $\tilde{\varphi}_r(X)$ be a Lagrange polynomial of degree d:

```
import sympy as sym
import numpy as np

def phi_r(r, X, d):
    if isinstance(X, sym.Symbol):
        h = sym.Rational(1, d)  # node spacing
        nodes = [2*i*h - 1 for i in range(d+1)]
    else:
        # assume X is numeric: use floats for nodes
        nodes = np.linspace(-1, 1, d+1)
    return Lagrange_polynomial(X, r, nodes)

def Lagrange_polynomial(x, i, points):
    p = 1
    for k in range(len(points)):
        if k != i:
            p *= (x - points[k])/(points[i] - points[k])
    return p

def basis(d=1):
    """Return the complete basis."""
    X = sym.Symbol('X')
    phi = [phi_r(r, X, d) for r in range(d+1)]
    return phi
```

Compute the element matrix

```
def element_matrix(phi, Omega_e, symbolic=True):
    n = len(phi)
    A_e = sym.zeros((n, n))
    X = sym.Symbol('X')
    if symbolic:
        h = sym.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    detJ = h/2  # dx/dX
    for r in range(n):
        for s in range(r, n):
            A_e[r,s] = sym.integrate(phi[r]*phi[s]*detJ, (X, -1, 1))
            A_e[s,r] = A_e[r,s]
    return A_e
```

Example on symbolic vs numeric element matrix

```
>>> from fe_approx1D import *
>>> phi = basis(d=1)
>>> phi
[1/2 - X/2, 1/2 + X/2]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=True)
[h/3, h/6]
[h/6, h/3]
>>> element_matrix(phi, Omega_e=[0.1, 0.2], symbolic=False)
[0.0333333333333333, 0.0166666666666667]
[0.0166666666666667, 0.0333333333333333]
```

Compute the element vector

```
def element_vector(f, phi, Omega_e, symbolic=True):
    n = len(phi)
    b_e = sym.zeros((n, 1))
    # Make f a function of X
    X = sym.Symbol('X')
    if symbolic:
        h = sym.Symbol('h')
    else:
        h = Omega_e[1] - Omega_e[0]
    x = (Omega_e[0] + Omega_e[1])/2 + h/2*X  # mapping
    f = f.subs('x', x)  # substitute mapping formula for x
    detJ = h/2  # dx/dX
    for r in range(n):
        b_e[r] = sym.integrate(f*phi[r]*detJ, (X, -1, 1))
    return b_e
```

Note `f.subs('x', x)`: replace `x` by $x(X)$ such that `f` contains `X`

Fallback on numerical integration if symbolic integration of $\int f\tilde{\varphi}_r dx$ fails

- Element matrix: only polynomials and sympy always succeeds
- Element vector: $\int f\tilde{\varphi}\,\mathrm{d}x$ can fail (sympy then returns an `Integral` object instead of a number)

```python
def element_vector(f, phi, Omega_e, symbolic=True):
        ...
        I = sym.integrate(f*phi[r]*detJ, (X, -1, 1))  # try...
        if isinstance(I, sym.Integral):
            h = Omega_e[1] - Omega_e[0]  # Ensure h is numerical
            detJ = h/2
            integrand = sym.lambdify([X], f*phi[r]*detJ)
            I = sym.mpmath.quad(integrand, [-1, 1])
        b_e[r] = I
        ...
```

Linear system assembly and solution

```python
def assemble(nodes, elements, phi, f, symbolic=True):
    N_n, N_e = len(nodes), len(elements)
    zeros = sym.zeros if symbolic else np.zeros
    A = zeros((N_n, N_n))
    b = zeros((N_n, 1))
    for e in range(N_e):
        Omega_e = [nodes[elements[e][0]], nodes[elements[e][-1]]]

        A_e = element_matrix(phi, Omega_e, symbolic)
        b_e = element_vector(f, phi, Omega_e, symbolic)

        for r in range(len(elements[e])):
            for s in range(len(elements[e])):
                A[elements[e][r],elements[e][s]] += A_e[r,s]
            b[elements[e][r]] += b_e[r]
    return A, b
```

Linear system solution

```python
if symbolic:
    c = A.LUsolve(b)           # sympy arrays, symbolic Gaussian elim.
else:
    c = np.linalg.solve(A, b)  # numpy arrays, numerical solve
```

Note: the symbolic computation of `A`, `b` and `A.LUsolve(b)` can be very tedious.

Example on computing symbolic approximations

```python
>>> h, x = sym.symbols('h x')
>>> nodes = [0, h, 2*h]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,    0]
[h/6, 2*h/3, h/6]
[  0,   h/6, h/3]
>>> b
[     h**2/6 - h**3/12]
[      h**2 - 7*h**3/6]
[5*h**2/6 - 17*h**3/12]
>>> c = A.LUsolve(b)
>>> c
[                           h**2/6]
[12*(7*h**2/12 - 35*h**3/72)/(7*h)]
[  7*(4*h**2/7 - 23*h**3/21)/(2*h)]
```

Example on computing numerical approximations

```python
>>> nodes = [0, 0.5, 1]
>>> elements = [[0, 1], [1, 2]]
>>> phi = basis(d=1)
>>> x = sym.Symbol('x')
>>> f = x*(1-x)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=False)
>>> A
[ 0.166666666666667, 0.0833333333333333,                  0]
[0.0833333333333333,  0.333333333333333, 0.0833333333333333]
[                 0, 0.0833333333333333,  0.166666666666667]
>>> b
[          0.03125]
[0.104166666666667]
[          0.03125]
>>> c = A.LUsolve(b)
>>> c
[0.0416666666666666]
[ 0.291666666666667]
[0.0416666666666666]
```

The structure of the coefficient matrix

```python
>>> d=1; N_e=8; Omega=[0,1]  # 8 linear elements on [0,1]
>>> phi = basis(d)
>>> f = x*(1-x)
>>> nodes, elements = mesh_symbolic(N_e, d, Omega)
>>> A, b = assemble(nodes, elements, phi, f, symbolic=True)
>>> A
[h/3,   h/6,     0,     0,     0,     0,     0,     0,   0]
[h/6, 2*h/3,   h/6,     0,     0,     0,     0,     0,   0]
[  0,   h/6, 2*h/3,   h/6,     0,     0,     0,     0,   0]
[  0,     0,   h/6, 2*h/3,   h/6,     0,     0,     0,   0]
[  0,     0,     0,   h/6, 2*h/3,   h/6,     0,     0,   0]
[  0,     0,     0,     0,   h/6, 2*h/3,   h/6,     0,   0]
[  0,     0,     0,     0,     0,   h/6, 2*h/3,   h/6,   0]
[  0,     0,     0,     0,     0,     0,   h/6, 2*h/3, h/6]
[  0,     0,     0,     0,     0,     0,     0,   h/6, h/3]
```

Note: do this by hand to understand what is going on!

General result: the coefficient matrix is sparse

- Sparse = most of the entries are zeros
- Below: P1 elements

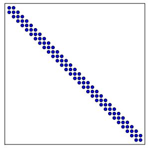
$$
A = \frac{h}{6}\left(\begin{array}{cccccccccc}
2 & 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
1 & 4 & 1 & \ddots & & & & & \vdots \\
0 & 1 & 4 & 1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & 1 & 4 & 1 & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & 1 & 4 & 1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 1 & 2
\end{array}\right)
$$

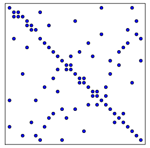
Exemplifying the sparsity for P2 elements

$$
A = \frac{h}{30}\left(\begin{array}{ccccccccc}
4 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 16 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 2 & 8 & 2 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 16 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 2 & 8 & 2 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 2 & 16 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 2 & 8 & 2 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 16 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 2 & 4
\end{array}\right)
$$

Matrix sparsity pattern for regular/random numbering of P1 elements

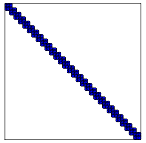
- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily

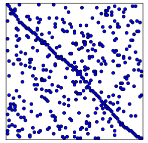




Matrix sparsity pattern for regular/random numbering of P3 elements

- Left: number nodes and elements from left to right
- Right: number nodes and elements arbitrarily





Sparse matrix storage and solution

The minimum storage requirements for the coefficient matrix $A_{i,j}$:

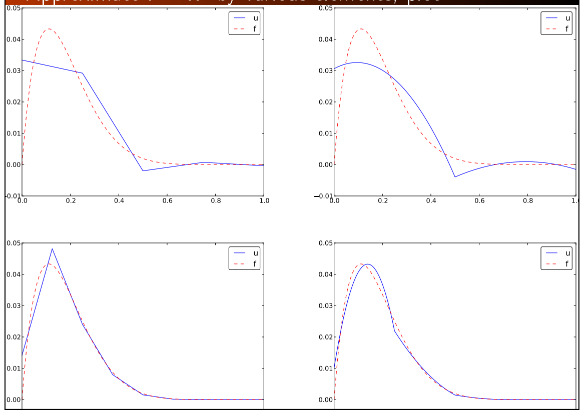
- P1 elements: only 3 nonzero entries per row
- P2 elements: only 5 nonzero entries per row
- P3 elements: only 7 nonzero entries per row
- It is important to utilize sparse storage and sparse solvers
- In Python: `scipy.sparse` package

Approximate $f \sim x^9$ by various elements; code

Compute a mesh with N_e elements, basis functions of degree d, and approximate a given symbolic expression $f(x)$ by a finite element expansion $u(x) = \sum_j c_j \varphi_j(x)$:

```
import sympy as sym
from fe_approx1D import approximate
x = sym.Symbol('x')

approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=4)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=2)
approximate(f=x*(1-x)**8, symbolic=False, d=1, N_e=8)
approximate(f=x*(1-x)**8, symbolic=False, d=2, N_e=4)
```

Approximate $f \sim x^9$ by various elements; plot



Comparison of finite element and finite difference approximation

- Finite difference approximation u_i of a function $f(x)$: simply choose $u_i = f(x_i)$
- This is the same as $u \approx \sum_i c_i\varphi_i$ + interpolation (see next slide)
- $u \approx \sum_i c_i\varphi_i$ + Galerkin/projection or least squares method: must derive and solve a linear system
- What is *really* the difference in the approximation u?

Interpolation/collocation with finite elements

Let $\{x_i\}_{i\in\mathcal{I}_s}$ be the nodes in the mesh. Collocation/interpolation means

$$u(x_i) = f(x_i), \quad i \in \mathcal{I}_s,$$

which translates to

$$\sum_{j\in\mathcal{I}_s} c_j\varphi_j(x_i) = f(x_i),$$

but $\varphi_j(x_i) = 0$ if $i \neq j$ so the sum collapses to one term $c_i\varphi_i(x_i) = c_i$, and we have the result

$$c_i = f(x_i)$$

Same result as the standard finite difference approach, but finite elements define u also *between* the x_i points

Galerkin/project and least squares vs collocation/interpolation or finite differences

- Scope: work with P1 elements
- Use projection/Galerkin or least squares (equivalent)
- Interpret the resulting linear system as finite difference equations

The P1 finite element machinery results in a linear system where equation no i is

$$\frac{h}{6}(u_{i-1} + 4u_i + u_{i+1}) = (f, \varphi_i)$$

Note:

- We have used u_i for c_i to make notation similar to finite differences
- The finite difference counterpart is just $u_i = f_i$

Expressing the left-hand side in finite difference operator notation

Rewrite the left-hand side of finite element equation no i:

$$h(u_i + \frac{1}{6}(u_{i-1} - 2u_i + u_{i+1})) = [h(u + \frac{h^2}{6}D_xD_x u)]_i$$

This is the standard finite difference approximation of

$$h(u + \frac{h^2}{6}u'')$$

Treating the right-hand side; Trapezoidal rule

$$(f, \varphi_i) = \int_{x_{i-1}}^{x_i} f(x)\frac{1}{h}(x - x_{i-1})dx + \int_{x_i}^{x_{i+1}} f(x)\frac{1}{h}(1 - (x - x_i))dx$$

Cannot do much unless we specialize f or use *numerical integration*.

Trapezoidal rule using the nodes:

$$(f, \varphi_i) = \int_\Omega f\varphi_i dx \approx h\frac{1}{2}(f(x_0)\varphi_i(x_0) + f(x_N)\varphi_i(x_N)) + h\sum_{j=1}^{N-1} f(x_j)\varphi_i(x_j)$$

$\varphi_i(x_j) = \delta_{ij}$, so this formula collapses to one term:

$$(f, \varphi_i) \approx hf(x_i), \quad i = 1, \ldots, N-1\,.$$

Same result as in collocation (interpolation) and the finite difference method!

Treating the right-hand side; Simpson's rule

$$\int_\Omega g(x)dx \approx \frac{h}{6}\left(g(x_0) + 2\sum_{j=1}^{N-1} g(x_j) + 4\sum_{j=0}^{N-1} g(x_{j+\frac{1}{2}}) + f(x_{2N})\right),$$

Our case: $g = f\varphi_i$. The sums collapse because $\varphi_i = 0$ at most of the points.

$$(f, \varphi_i) \approx \frac{h}{3}(f_{i-\frac{1}{2}} + f_i + f_{i+\frac{1}{2}})$$

Conclusions:

- While the finite difference method just samples f at x_i, the finite element method applies an average (smoothing) of f around x_i
- On the left-hand side we have a term $\sim hu''$, and u'' also contribute to smoothing
- There is some inherent smoothing in the finite element method

Finite element approximation vs finite differences

With Trapezoidal integration of (f, φ_i), the finite element method essentially solve

$$u + \frac{h^2}{6}u'' = f, \quad u'(0) = u'(L) = 0,$$

by the finite difference method

$$[u + \frac{h^2}{6}D_x D_x u = f]_i$$

With Simpson integration of (f, φ_i) we essentially solve

$$[u + \frac{h^2}{6}D_x D_x u = \bar{f}]_i,$$

where

$$\bar{f}_i = \frac{1}{3}(f_{i-1/2} + f_i + f_{i+1/2})$$

Note: as $h \rightarrow 0$, $hu'' \rightarrow 0$ and $\bar{f}_i \rightarrow f_i$.

Making finite elements behave as finite differences

- Can we adjust the finite element method so that we do not get the extra hu'' smoothing term and averaging of f?
- This allows finite elements to inherit (desired) properties of finite differences

Result:

- Compute all integrals by the Trapezoidal method and P1 elements
- Specifically, the coefficient matrix becomes diagonal ("lumped") - no linear system (!)
- Loss of accuracy? The Trapezoidal rule has error $\mathcal{O}(h^2)$, the same as the approximation error in P1 elements

Limitations of the nodes and element concepts

So far,

- *Nodes*: points for defining φ_i and computing u values
- *Elements*: subdomain (containing a few nodes)
- This is a common notion of nodes and elements

One problem:

- Our algorithms need nodes at the element boundaries
- This is often not desirable, so we need to throw the `nodes` and `elements` arrays away and find a more generalized element concept

The generalized element concept has cells, vertices, nodes, and degrees of freedom

- We introduce *cell* for the subdomain that we up to now called element
- A cell has *vertices* (interval end points)
- *Nodes* are, almost as before, points where we want to compute unknown functions
- *Degrees of freedom* is what the c_j represent (usually function values at nodes)

The concept of a finite element

1. a *reference cell* in a local reference coordinate system
2. a set of *basis functions* $\tilde{\varphi}_r$ defined on the cell
3. a set of *degrees of freedom* (e.g., function values) that uniquely determine the basis functions such that $\tilde{\varphi}_r = 1$ for degree of freedom number r and $\tilde{\varphi}_r = 0$ for all other degrees of freedom
4. a mapping between local and global degree of freedom numbers (*dof map*)
5. a geometric *mapping* of the reference cell onto to cell in the physical domain: $[-1, 1] \Rightarrow [x_L, x_R]$

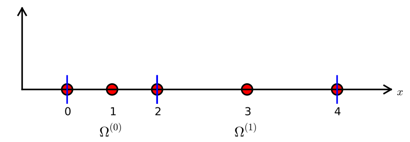
Basic data structures: vertices, cells, dof_map

- Cell vertex coordinates: `vertices` (equals `nodes` for P1 elements)
- Element vertices: `cells[e][r]` holds global vertex number of local vertex no `r` in element `e` (same as `elements` for P1 elements)
- `dof_map[e,r]` maps local dof `r` in element `e` to global dof number (same as `elements` for Pd elements)

The assembly process now applies `dof_map`:

```
A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
b[dof_map[e][r]] += b_e[r]
```

Example: data structures for P2 elements




```
vertices = [0, 0.4, 1]
cells = [[0, 1], [1, 2]]
dof_map = [[0, 1, 2], [2, 3, 4]]
```

Example: P0 elements

Example: Same mesh, but u is piecewise constant in each cell (P0 element). Same `vertices` and `cells`, but

```
dof_map = [[0], [1]]
```

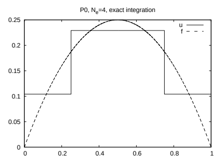
May think of one node in the middle of each element.

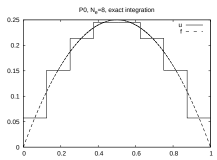
Note:
We will hereafter work with `cells`, `vertices`, and `dof_map`.

A program with the fundamental algorithmic steps

```python
# Use modified fe_approx1D module
from fe_approx1D_numint import *

x = sym.Symbol('x')
f = x*(1 - x)

N_e = 10
# Create mesh with P3 (cubic) elements
vertices, cells, dof_map = mesh_uniform(N_e, d=3, Omega=[0,1])

# Create basis functions on the mesh
phi = [basis(len(dof_map[e])-1) for e in range(N_e)]

# Create linear system and solve it
A, b = assemble(vertices, cells, dof_map, phi, f)
c = np.linalg.solve(A, b)

# Make very fine mesh and sample u(x) on this mesh for plotting
x_u, u = u_glob(c, vertices, cells, dof_map,
                resolution_per_element=51)
plot(x_u, u)
```

Approximating a parabola by P0 elements





The `approximate` function automates the steps in the previous slide:

```python
from fe_approx1D_numint import *
x=sym.Symbol("x")
for N_e in 4, 8:
    approximate(x*(1-x), d=0, N_e=N_e, Omega=[0,1])
```

Computing the error of the approximation; principles

$$L^2 \text{ error:} \quad ||e||_{L^2} = \left(\int_\Omega e^2 dx\right)^{1/2}$$

Accurate approximation of the integral:

- Sample $u(x)$ at many points in each element (call `u_glob`, returns `x` and `u`)
- Use the Trapezoidal rule based on the samples
- It is important to integrate u accurately *over the elements*
- (In a finite difference method we would just sample the mesh point values)

Computing the error of the approximation; details

Note

We need a version of the Trapezoidal rule valid for non-uniformly spaced points:

$$\int_\Omega g(x)dx \approx \sum_{j=0}^{n-1} \frac{1}{2}(g(x_j) + g(x_{j+1}))(x_{j+1} - x_j)$$

```
# Given c, compute x and u values on a very fine mesh
x, u = u_glob(c, vertices, cells, dof_map,
              resolution_per_element=101)
# Compute the error on the very fine mesh
e = f(x) - u
e2 = e**2
# Vectorized Trapezoidal rule
E = np.sqrt(0.5*np.sum((e2[:-1] + e2[1:])*(x[1:] - x[:-1]))
```

How does the error depend on h and d?

Theory and experiments show that the least squares or projection/Galerkin method in combination with Pd elements of equal length h has an error

$$||e||_{L^2} = Ch^{d+1}$$

where C depends on f, but not on h or d.

Cubic Hermite polynomials; definition

- Can we construct $\varphi_i(x)$ with continuous derivatives? Yes!

Consider a reference cell $[-1, 1]$. We introduce two nodes, $X = -1$ and $X = 1$. The degrees of freedom are

- 0: value of function at $X = -1$
- 1: value of first derivative at $X = -1$
- 2: value of function at $X = 1$
- 3: value of first derivative at $X = 1$

Derivatives as unknowns ensure the same $\varphi_i'(x)$ value at nodes and thereby continuous derivatives.

Cubic Hermite polynomials; derivation

4 constraints on $\tilde\varphi_r$ (1 for dof r, 0 for all others):

- $\tilde\varphi_0(X_{(0)}) = 1$, $\tilde\varphi_0(X_{(1)}) = 0$, $\tilde\varphi_0'(X_{(0)}) = 0$, $\tilde\varphi_0'(X_{(1)}) = 0$
- $\tilde\varphi_1'(X_{(0)}) = 1$, $\tilde\varphi_1'(X_{(1)}) = 0$, $\tilde\varphi_1(X_{(0)}) = 0$, $\tilde\varphi_1(X_{(1)}) = 0$
- $\tilde\varphi_2(X_{(1)}) = 1$, $\tilde\varphi_2(X_{(0)}) = 0$, $\tilde\varphi_2'(X_{(0)}) = 0$, $\tilde\varphi_2'(X_{(1)}) = 0$
- $\tilde\varphi_3'(X_{(1)}) = 1$, $\tilde\varphi_3'(X_{(0)}) = 0$, $\tilde\varphi_3(X_{(0)}) = 0$, $\tilde\varphi_3(X_{(1)}) = 0$

This gives 4 linear, coupled equations *for each* $\tilde\varphi_r$ to determine the 4 coefficients in the cubic polynomial

Cubic Hermite polynomials; result

$$\tilde\varphi_0(X) = 1 - \frac{3}{4}(X+1)^2 + \frac{1}{4}(X+1)^3 \quad (19)$$

$$\tilde\varphi_1(X) = -(X+1)(1 - \frac{1}{2}(X+1))^2 \quad (20)$$

$$\tilde\varphi_2(X) = \frac{3}{4}(X+1)^2 - \frac{1}{2}(X+1)^3 \quad (21)$$

$$\tilde\varphi_3(X) = -\frac{1}{2}(X+1)(\frac{1}{2}(X+1)^2 - (X+1)) \quad (22)$$

$$(23)$$

Numerical integration

- $\int_\Omega f\varphi_i dx$ must in general be computed by numerical integration
- Numerical integration is often used for the matrix too

Common form of a numerical integration rule

$$\int_{-1}^{1} g(X)dX \approx \sum_{j=0}^{M} w_j g(\bar{X}_j),$$

where

- $\bar{X}_j$ are *integration points*
- w_j are *integration weights*

Different rules correspond to different choices of points and weights

The Midpoint rule

Simplest possibility: the Midpoint rule,

$$\int_{-1}^{1} g(X)dX \approx 2g(0), \quad \bar{X}_0 = 0,\ w_0 = 2,$$

Exact for linear integrands

Newton-Cotes rules apply the nodes

- Idea: use a fixed, uniformly distributed set of points in $[-1, 1]$
- The points often coincides with nodes
- Very useful for making $\varphi_i \varphi_j = 0$ and get diagonal ("mass") matrices ("lumping")

The Trapezoidal rule:

$$\int_{-1}^{1} g(X)dX \approx g(-1) + g(1), \quad \bar{X}_0 = -1,\ \bar{X}_1 = 1,\ w_0 = w_1 = 1,$$

Simpson's rule:

$$\int_{-1}^{1} g(X)dX \approx \frac{1}{3}\left(g(-1) + 4g(0) + g(1)\right),$$

where

$$\bar{X}_0 = -1,\ \bar{X}_1 = 0,\ \bar{X}_2 = 1,\ w_0 = w_2 = \frac{1}{\,},\ w_1 = \frac{4}{\,}$$

Gauss-Legendre rules apply optimized points

- Optimize the location of points to get higher accuracy
- Gauss-Legendre rules (quadrature) adjust points and weights to integrate polynomials exactly

$$M = 1: \quad \bar{X}_0 = -\frac{1}{\sqrt{3}},\ \bar{X}_1 = \frac{1}{\sqrt{3}},\ w_0 = w_1 = 1 \tag{24}$$

$$M = 2: \quad \bar{X}_0 = -\sqrt{\frac{3}{5}},\ \bar{X}_0 = 0,\ \bar{X}_2 = \sqrt{\frac{3}{5}},\ w_0 = w_2 = \frac{5}{9},\ w_1 = \frac{8}{9} \tag{25}$$

- $M = 1$: integrates 3rd degree polynomials exactly
- $M = 2$: integrates 5th degree polynomials exactly
- In general, M-point rule integrates a polynomial of degree $2M + 1$ exactly.

See `numint.py` for a large collection of Gauss-Legendre rules.

Approximation of functions in 2D

Extensibility of 1D ideas.

All the concepts and algorithms developed for approximation of 1D functions $f(x)$ can readily be extended to 2D functions $f(x, y)$ and 3D functions $f(x, y, z)$. Key formulas stay the same.

Quick overview of the 2D case

Inner product in 2D:

$$(f, g) = \int_{\Omega} f(x, y)g(x, y)dxdy$$

Least squares and project/Galerkin lead to a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s$$

$$A_{i,j} = (\psi_i, \psi_j)$$

$$b_i = (f, \psi_i)$$

Challenge: How to construct 2D basis functions $\psi_i(x, y)$?

2D basis functions as tensor products of 1D functions

Use a 1D basis for x variation and a similar for y variation:

$$V_x = \text{span}\{\hat{\psi}_0(x), \ldots, \hat{\psi}_{N_x}(x)\} \tag{26}$$
$$V_y = \text{span}\{\hat{\psi}_0(y), \ldots, \hat{\psi}_{N_y}(y)\} \tag{27}$$

The 2D vector space can be defined as a *tensor product* $V = V_x \otimes V_y$ with basis functions

$$\psi_{p,q}(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y) \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y\,.$$

Tensor products

Given two vectors $a = (a_0, \ldots, a_M)$ and $b = (b_0, \ldots, b_N)$ their *outer tensor product*, also called the *dyadic product*, is $p = a \otimes b$, defined through

$$p_{i,j} = a_i b_j, \quad i = 0, \ldots, M,\ j = 0, \ldots, N\,.$$

Note: p has two indices (as a matrix or two-dimensional array)

Example: 2D basis as tensor product of 1D spaces,

$$\psi_{p,q}(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y), \quad p \in \mathcal{I}_x, q \in \mathcal{I}_y$$

Double or single index?

The 2D basis can employ a double index and double sum:

$$u = \sum_{p\in\mathcal{I}_x}\sum_{q\in\mathcal{I}_y} c_{p,q}\psi_{p,q}(x,y)$$

Or just a single index:

$$u = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x,y)$$

with an index mapping $(p,q) \rightarrow i$:

$$\psi_i(x,y) = \hat{\psi}_p(x)\hat{\psi}_q(y), \quad i = p(N_y+1)+q \text{ or } i = q(N_x+1)+p$$

Example on 2D (bilinear) basis functions; formulas

In 1D we use the basis

$$\{1, x\}$$

2D tensor product (all combinations):

$$\psi_{0,0} = 1, \quad \psi_{1,0} = x, \quad \psi_{0,1} = y, \quad \psi_{1,1} = xy$$

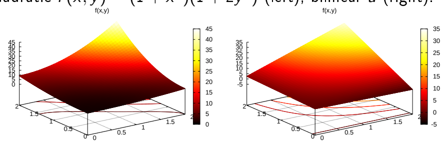
or with a single index:

$$\psi_0 = 1, \quad \psi_1 = x, \quad \psi_2 = y, \quad \psi_3 = xy$$

See notes for details of a hand-calculation.

Example on 2D (bilinear) basis functions; plot

Quadratic $f(x,y) = (1+x^2)(1+2y^2)$ (left), bilinear u (right):



Implementation; principal changes to the 1D code

Very small modification of `approx1D.py`:

- `Omega = [[0, L_x], [0, L_y]]`
- Symbolic integration in 2D
- Construction of 2D (tensor product) basis functions

Implementation; 2D integration

```
import sympy as sym

integrand = psi[i]*psi[j]
I = sym.integrate(integrand,
                  (x, Omega[0][0], Omega[0][1]),
                  (y, Omega[1][0], Omega[1][1]))

# Fall back on numerical integration if symbolic integration
# was unsuccessful
if isinstance(I, sym.Integral):
    integrand = sym.lambdify([x,y], integrand)
    I = sym.mpmath.quad(integrand,
                        [Omega[0][0], Omega[0][1]],
                        [Omega[1][0], Omega[1][1]])
```

Implementation; 2D basis functions

Tensor product of 1D "Taylor-style" polynomials x^i:

```
def taylor(x, y, Nx, Ny):
    return [x**i*y**j for i in range(Nx+1) for j in range(Ny+1)]
```

Tensor product of 1D sine functions $\sin((i+1)\pi x)$:

```
def sines(x, y, Nx, Ny):
    return [sym.sin(sym.pi*(i+1)*x)*sym.sin(sym.pi*(j+1)*y)
            for i in range(Nx+1) for j in range(Ny+1)]
```

Complete code in `approx2D.py`

Implementation; application

$f(x, y) = (1 + x^2)(1 + 2y^2)$

```
>>> from approx2D import *
>>> f = (1+x**2)*(1+2*y**2)
>>> psi = taylor(x, y, 1, 1)
>>> Omega = [[0, 2], [0, 2]]
>>> u, c = least_squares(f, psi, Omega)
>>> print u
8*x*y - 2*x/3 + 4*y/3 - 1/9
>>> print sym.expand(f)
2*x**2*y**2 + x**2 + 2*y**2 + 1
```

Implementation; trying a perfect expansion

Add higher powers to the basis such that $f \in V$:

```
>>> psi = taylor(x, y, 2, 2)
>>> u, c = least_squares(f, psi, Omega)
>>> print u
2*x**2*y**2 + x**2 + 2*y**2 + 1
>>> print u-f
0
```

Expected: $u = f$ when $f \in V$

Generalization to 3D

Key idea:

$$V = V_x \otimes V_y \otimes V_z$$

Repeated outer tensor product of multiple vectors

$$a^{(q)} = (a_0^{(q)}, \ldots, a_{N_q}^{(q)}), \quad q = 0, \ldots, m$$
$$p = a^{(0)} \otimes \cdots \otimes a^{(m)}$$
$$p_{i_0, i_1, \ldots, i_m} = a_{i_1}^{(0)} a_{i_1}^{(1)} \cdots a_{i_m}^{(m)}$$

$$\psi_{p,q,r}(x, y, z) = \hat{\psi}_p(x)\hat{\psi}_q(y)\hat{\psi}_r(z)$$
$$u(x, y, z) = \sum_{p\in\mathcal{I}_x} \sum_{q\in\mathcal{I}_y} \sum_{r\in\mathcal{I}_z} c_{p,q,r}\psi_{p,q,r}(x, y, z)$$

Finite elements in 2D and 3D

The two great advantages of the finite element method:

- Can handle complex-shaped domains in 2D and 3D
- Can easily provide higher-order polynomials in the approximation

Finite elements in 1D: mostly for learning, insight, debugging

Examples on cell types

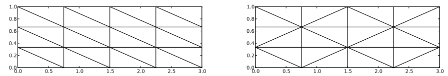
2D:

- triangles
- quadrilaterals

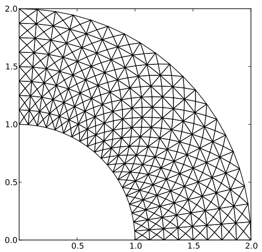
3D:

- tetrahedra
- hexahedra

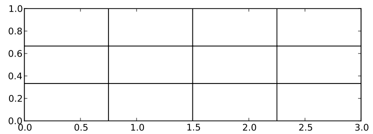
Rectangular domain with 2D P1 elements



Deformed geometry with 2D P1 elements

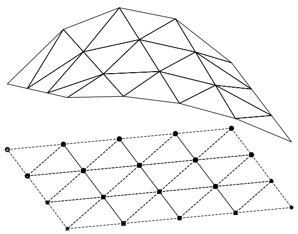


Rectangular domain with 2D Q1 elements



Basis functions over triangles in the physical domain

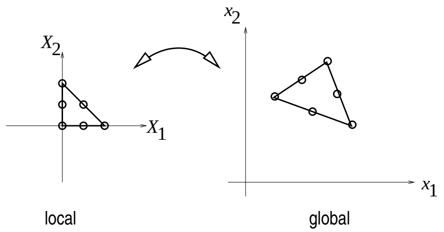
The P1 triangular 2D element: u is linear $ax + by + c$ over each triangular cell



Basic features of 2D elements

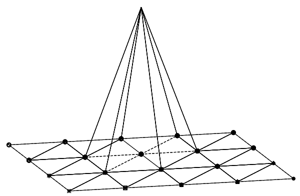
- Cells = triangles
- Vertices = corners of the cells
- Nodes = vertices
- Degrees of freedom = function values at the nodes

Linear mapping of reference element onto general triangular cell



φ_i: pyramid shape, composed of planes

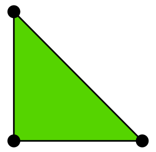
- $\varphi_i(x, y)$ varies linearly over each cell
- $\varphi_i = 1$ at vertex (node) i, 0 at all other vertices (nodes)



Element matrices and vectors

- As in 1D, the contribution from one cell to the matrix involves just a few entries, collected in the element matrix and vector
- $\varphi_i\varphi_j \neq 0$ only if i and j are degrees of freedom (vertices/nodes) in the same element
- The 2D P1 element has a 3×3 element matrix

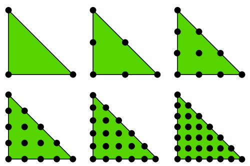
Basis functions over triangles in the reference cell



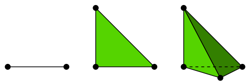
$$\tilde{\varphi}_0(X, Y) = 1 - X - Y \tag{28}$$
$$\tilde{\varphi}_1(X, Y) = X \tag{29}$$
$$\tilde{\varphi}_2(X, Y) = Y \tag{30}$$

Higher-degree $\tilde{\varphi}_r$ introduce more nodes (dof = node values)

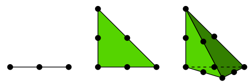
2D P1, P2, P3, P4, P5, and P6 elements



P1 elements in 1D, 2D, and 3D



P2 elements in 1D, 2D, and 3D



- Interval, triangle, tetrahedron: *simplex* element (plural quick-form: *simplices*)
- Side of the cell is called *face*
- Thetrahedron has also *edges*

Affine mapping of the reference cell; formula

Mapping of local $\boldsymbol{X} = (X, Y)$ coordinates in the reference cell to global, physical $\boldsymbol{x} = (x, y)$ coordinates:

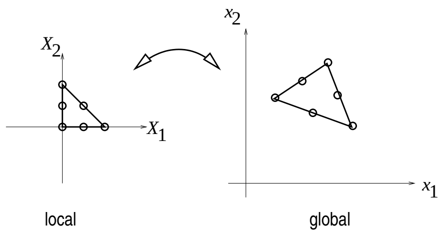
$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r^{(1)}(\boldsymbol{X})\boldsymbol{x}_{q(e,r)} \tag{31}$$

where

- r runs over the local vertex numbers in the cell
- $\boldsymbol{x}_i$ are the (x, y) coordinates of vertex i
- $\tilde{\varphi}_r^{(1)}$ are P1 basis functions

This mapping preserves the straight/planar faces and edges.

Affine mapping of the reference cell; figure

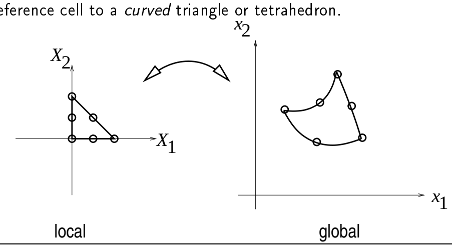


Isoparametric mapping of the reference cell

Idea: Use the basis functions of the element (not only the P1 functions) to map the element

$$\boldsymbol{x} = \sum_r \tilde{\varphi}_r(\boldsymbol{X})\boldsymbol{x}_{q(e,r)}$$

Advantage: higher-order polynomial basis functions now map the reference cell to a *curved* triangle or tetrahedron.



Computing integrals

Integrals must be transformed from $\Omega^{(e)}$ (physical cell) to $\tilde{\Omega}^r$ (reference cell):

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})\varphi_j(\boldsymbol{x})\,\mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})\tilde{\varphi}_j(\boldsymbol{X})\det J\,\mathrm{d}X \tag{32}$$

$$\int_{\Omega^{(e)}} \varphi_i(\boldsymbol{x})f(\boldsymbol{x})\,\mathrm{d}x = \int_{\tilde{\Omega}^r} \tilde{\varphi}_i(\boldsymbol{X})f(\boldsymbol{x}(\boldsymbol{X}))\det J\,\mathrm{d}X \tag{33}$$

where $\mathrm{d}x = dxdy$ or $\mathrm{d}x = dxdydz$ and $\det J$ is the determinant of the Jacobian of the mapping $\boldsymbol{x}(\boldsymbol{X})$.

$$J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \det J = \frac{\partial x}{\partial X}\frac{\partial y}{\partial Y} - \frac{\partial x}{\partial Y}\frac{\partial y}{\partial X}$$

Affine mapping (31): $\det J = 2\Delta$, $\Delta =$ cell volume

Remark on going from 1D to 2D/3D

Finite elements in 2D and 3D builds on the same *ideas* and *concepts* as in 1D, but there is simply much more to compute because the specific mathematical formulas in 2D and 3D are more complicated and the book keeping with dof maps also gets more complicated. The manual work is tedious, lengthy, and error-prone so automation by the computer is a must.